Lab 4 asked of us to do many things. To compute the height of a B-tree, extract the items of the B-tree into a sorted list, return the minimum element at depth d, the maximum at depth d, number of nodes at depth d, print all items at depth d. To return the number of nodes that are full, number leaves that are full, and given a number k return the depth at which k is found.

The very first thing asked of us to do was to return the height of a b-tree. This is pretty simple to do since the height of the tree is the same throughout the entire tree so you can go down to any leaf node and return the height of the leave node. That was the idea and to do it I just created a while loop that ran until it reached a leaf node adding 1 to a depth counter.

The next thing asked was to extract the items into a sorted list. To do this I would create a base case of if T was a leaf to return t.item. If t was not a leaf, I would then create an empty native list and add a looped recursive call that would first add the recursive calls to the list then add the next item currently in t, then loop the case until T has no more children. Afterwards you return the list that has have everything added into it.

To return the minimum element at depth d I would recursively call the function taking one off of d each call and if d was 0 then return the zero element in t.item, if T is a leaf and d still is not zero then I would return math.inf. Returning the maximum element is the same idea but I returned the -1 item taking advantage of the way native python lists function.

After those we were asked to return the number of nodes at depth d, in this problem we have to visit every node until we reach depth d. The idea is simple, if d is zero then we return 1, if not then if t is a leaf and d is not zero then we return -1 since no nodes exist past the leaf nodes. If neither case is true then we loop the recursive call storing what they return in a variable, If for some reason the function returned -1 then we just return -1. Other wise we continue to add 1 per recursive call returning the variable at the end of the function.

To print all the items at a depth d, if d is zero then you print all the items in t.item, if it’s a leaf and d is not zero you return. If neither then you loop a recursive call visiting every node until d is reached, because we need to print every item at depth d.

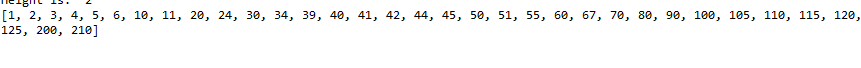
The function to find the number of full nodes didn’t give much trouble either. All you have to do check if the length of t.item is equal to the max items value held within the B-tree class. If so, you return 1. If not and t is a leaf, then return zero. If neither is true, we create a variable and save recursive calls return values into it. At the end we return that variable that has added all the number of full nodes. The next function that checks to number of full leaf nodes works the same. The only difference is that in the base case when it finds a full node, it checks if it is then a leaf node before returning the value of one or zero.

The final part asked of us was to return the depth that the item k is at. This was the most tricky of them all but not that hard. The first thing we must do is see if k is in the current T.item. If it is there then we just return zero, if not and t is a leaf node then we return -1. If neither is true we locate where k will be in the tree, if it’s greater than the largest item in T then we go to the last child. If not we see where it would be by going through each item seeing If k is less than each item and when it is we recursively call the function saving the return value into d adding one per call, if k is never found then we return -1 to signify that the item k is not in the list.

Testing the code for problem one was easy, I would create different sized tree and see if it returned the correct depth. Using a B-tree that had a height of 2, the first being 0, the next depth 1 then depth 2. So in all the B-tree has 3 layers of nodes. This piece of code has a quick runtime only going through a constant time worth of data to find the layer giving it a practice runtime of O(1) as the number of calls to find the depth is always extremely low.

To extract the items into a list I would test it by printing the list returned by the function, if it printed every item that was in the B-tree in ascending order then it works. Given a b-tree that was given values in this order

[30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6, 24, 67, 44, 125, 210, 55, 51, 41,42, 34, 39]

The program would return the following list.

This block of code would is having a slower runtime since it does have to visit every single node to properly work, meaning it has a O(n) runtime.

 To return the minimum and maximum element at depth d you would always go down to depth d and then return the zero or -1 element respectively. Testing the code was to return the minimum and maximum value at various depths.

The output there is using the same b-tree as before. Like the height of the tree this runs in constant time always running d number of times or just the height of the tree which is always such a low number we consider it to be O(1) time.

Returning the number of Nodes was simple in testing, after drawing out the tree, and seeing what it ‘looks like’ we see if the number returned is correct, and to test we just use various types of b-trees with different numbers of nodes, if it returns the correct number of nodes then it works, which after testing my code consistently returns the correct number of nodes.

To print all items at depth d I would try the same thing and test different depths to see if it does print all the items correctly, the output my program would give with the b tree values stated before is the following. 

The output is correct as it prints at depth 2. The program has roughly log n run time as it doesn’t need to visit every node or do anything with nodes that aren’t at depth d

Number of Full nodes and Full leaf nodes were tested in the same way as well by visually making the b-tree and seeing if the program would output it correctly as well. To properly test I made one full node an internal node and another a leaf node. The number of full nodes should then be 2 and full leaf nodes be 1. Which after running the code shows that the functions properly work as shown below.



The final function to find the depth of item k The function initially upon testing wasn’t working for some reason, after a few more trials I found it was because my base case wasn’t done properly and I added the case at the very end to return 1 + d which solved my issue. After checking with various numbers both in and not in the tree I saw it was consistently outputting the correct value saying the correct depth for the item being searched for. Two example outputs can be found below.

This lab taught me how to better handle and manipulate b-trees. I was a bit nervous at first but this wasn’t too bad after taking some time to learn the material.

Appendix—

# -\*- coding: utf-8 -\*-

"""

"""

import math

class BTree(object):

# Constructor

def \_\_init\_\_(self,item=[],child=[],isLeaf=True,max\_items=5):

self.item = item

self.child = child

self.isLeaf = isLeaf

if max\_items <3: #max\_items must be odd and greater or equal to 3

max\_items = 3

if max\_items%2 == 0: #max\_items must be odd and greater or equal to 3

max\_items +=1

self.max\_items = max\_items

def FindChild(T,k):

# Determines value of c, such that k must be in subtree T.child[c], if k is in the BTree

for i in range(len(T.item)):

if k < T.item[i]:

return i

return len(T.item)

def InsertInternal(T,i):

# T cannot be Full

if T.isLeaf:

InsertLeaf(T,i)

else:

k = FindChild(T,i)

if IsFull(T.child[k]):

m, l, r = Split(T.child[k])

T.item.insert(k,m)

T.child[k] = l

T.child.insert(k+1,r)

k = FindChild(T,i)

InsertInternal(T.child[k],i)

def Split(T):

#print('Splitting')

#PrintNode(T)

mid = T.max\_items//2

if T.isLeaf:

leftChild = BTree(T.item[:mid])

rightChild = BTree(T.item[mid+1:])

else:

leftChild = BTree(T.item[:mid],T.child[:mid+1],T.isLeaf)

rightChild = BTree(T.item[mid+1:],T.child[mid+1:],T.isLeaf)

return T.item[mid], leftChild, rightChild

def InsertLeaf(T,i):

T.item.append(i)

T.item.sort()

def IsFull(T):

return len(T.item) >= T.max\_items

def Insert(T,i):

if not IsFull(T):

InsertInternal(T,i)

else:

m, l, r = Split(T)

T.item =[m]

T.child = [l,r]

T.isLeaf = False

k = FindChild(T,i)

InsertInternal(T.child[k],i)

def height(T):

if T.isLeaf:

return 0

return 1 + height(T.child[0])

def Search(T,k):

# Returns node where k is, or None if k is not in the tree

if k in T.item:

return T

if T.isLeaf:

return None

return Search(T.child[FindChild(T,k)],k)

def Print(T):

# Prints items in tree in ascending order

if T.isLeaf:

for t in T.item:

print(t,end=' ')

else:

for i in range(len(T.item)):

Print(T.child[i])

print(T.item[i],end=' ')

Print(T.child[len(T.item)])

def PrintD(T,space):

# Prints items and structure of B-tree

if T.isLeaf:

for i in range(len(T.item)-1,-1,-1):

print(space,T.item[i])

else:

PrintD(T.child[len(T.item)],space+' ')

for i in range(len(T.item)-1,-1,-1):

print(space,T.item[i])

PrintD(T.child[i],space+' ')

def SearchAndPrint(T,k):

node = Search(T,k)

if node is None:

print(k,'not found')

else:

print(k,'found',end=' ')

print('node contents:',node.item)

def getheight(T):

h = 0

while T.isLeaf is not True:

T = T.child[0]

h +=1

return h

def Btree\_ToList(T):

if T.isLeaf:#returns the leaf list to concatenate later

return T.item

NList = []#list we will return

for i in range(len(T.child)):

NList += Btree\_ToList(T.child[i])

if i< len(T.item):

NList.append(T.item[i])

return NList

def MinAtD(T, d):

if d == 0:

return T.item[0]

if T.isLeaf:

return -math.inf

return MinAtD(T.child[0], d-1)

def MaxAtD(T, d):

if d == 0:

return T.item[-1]

if T.isLeaf:

return -math.inf

return MaxAtD(T.child[-1],d-1)

def NumNodesAtD(T, d):

if d == 0:

return 1

if T.isLeaf:

return -1

NumNodes = 0

for i in range(len(T.child)):

NumNodes += NumNodesAtD(T.child[i], d-1)

if NumNodes < 0:

return -1

return NumNodes

def PrintNodesAtD(T, d):

if d == 0:

print(T.item, end = '')

if T.isLeaf:

return

for i in range(len(T.child)):

PrintNodesAtD(T.child[i], d-1)

def numFullNodes(T):

if len(T.item) == T.max\_items:

return 1

if T.isLeaf:

return 0

numMax =0

for i in range(len(T.child)):

numMax += numFullNodes(T.child[i])

return numMax

def numFullLeafs(T):

if len(T.item) == T.max\_items:

if T.isLeaf:

return 1

else:

return 0

numMax =0

for i in range(len(T.child)):

numMax += numFullLeafs(T.child[i])

return numMax

def depthOfK(T, k):

if k in T.item:

return 0

if T.isLeaf:

return -1#base cases

if k > T.item[-1]:

d = depthOfK(T.child[-1], k)

if d == -1:

return -1

else:

return 1+d

for i in range(len(T.item)):

if k < T.item[i]:

d = depthOfK(T.child[i], k)

if d == 0:

return d+1

if d < 0:

return -1

return 1 + d

L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6, 24, 67, 44, 125, 210, 55, 51, 41,42, 34, 39]

T = BTree()

for i in L:

Insert(T,i)

PrintD(T,'')

Print(T)

print('\n#####################################################')

print('Height is: ', getheight(T))

print(Btree\_ToList(T), '\n\n')

print(NumNodesAtD(T, 1))

PrintNodesAtD(T, 2)

print()

print('Min at 1:',MinAtD(T, 1))

print('Max at 1:',MaxAtD(T, 1))

print('Num full nodes: ', numFullNodes(T))

print('Num full leafs: ', numFullLeafs(T))

print('searching for item 500: ', depthOfK(T, 500))

“I certify that this project is entirely my own work. I wrote, debugged and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.”

- Seth Abel Flores